I'm not a bot

Rails form boolean radio button

In this demonstration, we'll explore how to pass data collected from a form in the view to the controller action for processing. We'll continue building on our QuizMe project, focusing on creating a Contact page that will be used to collect user feedback. We'll start by adding a new Contact page with the URL ` . This involves adding a new route, a new controller action (contact), and a new view (contact.html.erb). The contact.html.erb view should begin with the following base content: Contact Us We welcome your feedback. Please submit the form below to let us know what you think or alert us to any problems you are having with QuizMe.

Feedback

Next, we'll add a link to the Contact page at the bottom of the Welcome page, allowing users to access it. For this part, we'll create the feedback form depicted in Figure 1. The form will collect user information, including name, email, and whether QuizMe is allowed to contact them through their provided email address. Additionally, we'll include fields for the kind of feedback desired and a message to leave. Initially, users will be able to fill out the form as shown in Figure 2, but the app won't yet handle submissions. We'll create a new POST route for the URL ` that points to a new `leave_feedback` action in the StaticPagesController class and name it 'leave_feedback'. The new route should look like this: post 'contact', to: 'static_pages#leave_feedback', as: 'leave_feedback' We'll also create the `leave_feedback` action in the StaticPagesController class, which should return an HTML response that renders the contact.html.erb view. The action should resemble this: def leave_feedback respond_to do |format| format.html { render :contact } end end Finally, we'll replace the form placeholder comment in the contact.html.erb view with a form block using the `form_with` helper. that should send a POST request to th eleave_feedback_path URL by settin' the url and method arguments of form_with. The form_with block expect't an HTML response, so we use the local: true argument. Thus, th eform block should look like this: ... To add a text field for th eusers' name, we us e and . Th evalue in th efield should be set to nil at first. We can also use a div HTML element to group th eform fields together. Similarly, we add an email field for th eusers' email with and . For th econtact method, we create a radio button group with two options: yes or no. We us e fo ryes and fo rno. Each option should have its own radio_button_tag that controls the value passed back when th eform is submitted, along with a label_tag that specifies th estring th euser sees for each option. Note: I kept th eoriginal text's tone and language to maintain consistency. The options for radio buttons include the field ID, value returned when submitted, and a boolean indicating whether the button is checked. For this case, we want 'Yes' as the default option, so set it initially checked and the other not. Radio button groups require unique IDs in the label_tag options, with the display value not needing to match the return value. For feedback, add a dropdown list allowing users to choose between leaving a review or reporting a bug. The code for this should resemble: ```html ``` This dropdown list functions similarly to radio buttons, allowing users to select one option from a predefined set. Unlike the Rails helper for radio buttons, this helper includes all options by passing an array of values into the `options_for_select` function. The default text in the dropdown is set using the prompt option. Add a text area for users' messages: ```html
``` An HTML text area allows multi-line user input, similar to a text field. The options for this helper include the field ID ("message"), initial value (nil at first), and an optional size option. By default, the text area can hold 2 rows of 21 characters, but you can change these values by setting individual row or column options, or using the size option to specify both. A submit button for the feedback form should be created with the code . This generates an HTML input tag that looks like a button when rendered. The main argument of submit_tag is the text displayed on the button ("Send Feedback"). To process user data submitted via the form, the leave_feedback controller action should check if all required fields have values other than nil or blank strings. The params hash stores field IDs and values. A boolean variable form_complete can be set to true by iterating over each required field key in the params hash and checking that it exists with a non-blank value. The code for this is: required = [:name, :email, :reply, :feedback_type, :message] form_complete = true required.each do |f| if params.has_key? f and not params[f].blank? # good news, leave blank else form_complete = false end end the blank? method checks for nil or empty string values, then app determines form completeness, passes status msg to view. setup involves setting form_status_msg in controller based on form_complete boolean, and passing it into response. contact.html.erb displays status message if set. however, current app requires re-entering all fields when told form was incomplete. we can fix this by passing params hash back into view as feedback variable, then using that to set field values in contact.html.erb, falling back to default values if none provided in feedback hash. we might try to set this up differently using a ternary operator with the following syntax: however, making changes like this for every field could get complicated. instead, we can use hash fetch to get the value of a key if it exists or specify a return value if it doesn't exist. so, the updated code for text fields and text areas would be: for radio buttons, we need to ensure that the checked value is set based on the value for the "reply" key and convert true/false values to strings using the to_s method. so, it would look like this: for dropdown selects, the selected option is the second parameter to options_for_select. if nothing is selected, it should be set to an empty string, like this: currently, we have a working form that users can use to easily enter their feedback. but there's still room for improvement. upcoming demos will cover using remote form submission and flash messages. additionally, we'll see how to save and process the submitted data in future demos. `def getResults` if `params[:filter]` == 'month' @results = Multicasts.where(:extract(month from datetime) = ? , :month=>MONTH(date)) end if `params[:filter]` ==‘multi’ @results = Multicasts.where(:when => ‘im’) end if `params[:filter]` ==‘uni’ @results = Multicasts.where(:sms_voice => “sms”") end
```